ALGONQUIN COLLEGE

ASSIGNMENT 2 SPECIFICATION[1] - SCANNER

General View

Due Date: prior or on **March 20th 2021 (midnight)**

- **1st week late submission** (until March 27th midnight): 30% off.
- **2nd week late submission** (until April 6th midnight): 50% off.
- **Later:** 100% off.

Earnings: 20% of your course grade (plus 1% bonus)

Development: Activity can be done **individually** or in teams (**only 2 students** allowed).

Purpose: Development of a Scanner, using dynamic definition, RE (Regular Expressions) and FDA (Finite Deterministic Automata) implementation.

- This is an important activity from front-end compiler, and it will use several advanced datatypes, as well as function pointer in C coding style, incrementing the concepts used in programming techniques, data types and structures, memory management, and simple file input/output. The activity will also use the **buffer** previously defined by students. This assignment will be also an exercise in "*excessively defensive programming*".
- You are going to write functions that are required for the compiler's front-end and should be used by **parser** to identify tokens and will use a dynamic way to recognize tokens using tables and functions. To complete the assignment, you should fulfill some tasks presented later.
- The current version of code requires *Camel Code style*. Use it appropriately.

[1] Adapted from resources developed by Prof. Svillen Ranev (Algonquin College, 2019)

Task 1: RE, TD and TT for PLATYPUS (5 marks)

Do the activity described in **RETDTT_W21** document (check inside **Assignment 2.zip**). In this part you have to write **regular expressions** for AVID (Arithmetic Variable Identifiers), SVID (String Variable Identifiers), IL (Integer Literals). FPL (Float Point Literals), SL (String Literals), design the TD (**Transition Diagram**) and complete the TT (**Transition Table**).

- In this course you will have a gratifying experience to write the front-end of a compiler for a programming language named ***PLATYPUS*** (or, simply "PLATYPUS").
    - The PLATYPUS informal language specification is given in **PlatypusILS_W21** document.
    - The PLATYPUS formal specification (detailing **Grammar** and **BNF**) is given in **PlatypusBNFGR_W21**.
- In order to write a compiler, the informal language specification must be converted to a formal language specification.
    - Since PLATYPUS is a simple, yet complete, programming language it can be described formally with a context-free grammar (**BNF**) notation.
- The PLATYPUS grammar has two parts: a lexical grammar and a syntactic grammar.
    - The **lexical grammar** will define the lexical part of the language: the character set and the input elements such as white space, comments, and tokens. In Part 2 of the assignment, you will use the lexical grammar to implement a lexical analyzer (scanner).
    - The **syntactic grammar** has the tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions.
        - The productions, starting from the start symbol <program>, describe how a sequence of tokens can form syntactically correct PLATYPUS statements and programs.
        - This part of the grammar will be used to implement a syntax analyzer (parser) in one of the following assignments.
- You will find the complete lexical and syntactical grammar for the PLATYPUS language in the document **PlatypusBNFGR_W21**. Read very carefully the informal language specification (**PlatypusILS_W21**) and then see how the informal language specification has been converted to a formal specification using a BNF grammar notation.
- Part of the Scanner will be implemented using a Deterministic Finite Automaton (DFA) based on a Transition Table (TT) (see **Part 2** of this document).
    - The TT is partially given to you in file **CST8152_RETDTT_W21.doc** (model task). You need to complete it and include in your code (see table.h). To do this, you must:
        - Convert the lexical grammar into RE (Regular Expressions).

- Then using the regular expressions, you must draw a Transition Diagram (TD).
- Finally, using the Transition Diagram you can create the TT (Transition Table) that must be competed in your **CST8152_RETDTT_W21.doc** and code (table.h).

Task 2: Scanner Implementation (15 marks)

2.1. GENERAL VIEW

In Part 1, you had analyzed the grammar for the PLATYPUS programming language. Now, in Part 2, you need to create a lexical analyzer (**scanner**) for the PLATYPUS programming language.

Note 1: Remembering Scanner

*The scanner reads a source program from a text file and produces a stream of token representations. Actually, the scanner does not need to recognize and produce all the tokens before next phase of the compilation (the parsing) takes action. That is why, in almost all compilers, the scanner is actually a function that recognizes language tokens and produces token representations one at a time when called by the syntax analyzer (the **parser**).*

- In your implementation, the input to the lexical analyzer is a source program written in PLATYPUS language and seen as a stream of characters (symbols) loaded into an input buffer.
  - The output of each call to the Scanner is a single Token, to be requested and used, in a later assignment, by the Parser.
  - You need to use a data structure to represent the Token.
  - Your scanner will be a mixture between **token-driven scanner** and **transition-table driven** (DFA)
    - In **token-driven scanner** you have to write code for every token recognition.
    - **Transition-table driven scanners** are easy to implement with scanner generators.
  - The token is processed as a separate exceptional case (exception- or case-driven scanners).
  - They are difficult for modifications and maintenance (but in some cases could be faster and more efficient).

- **Transition-table driven part** of your scanner is to recognize only variable identifiers (including keywords), both arithmetical or string, integer literals (decimal constants), floating-point literals, and string literals.
    - To build transition table for those tokens you have to transform their grammar definitions into regular expressions and create the corresponding transition diagram(s) and transition table.
    - As you already know, **Regular Expressions** are a convenient means of specifying (describing) a set of strings.

2.2. IMPLEMENTATION OVERVIEW

- In Part 2, your task is to write a scanner program (set of functions). Three files are provided for you on Brightspace LMS: **token.h**, **table.h**, and **scanner.c** (see **Assignment 2.zip**). Where required, you must write a C code that provides the specified functionality. Your scanner program (project) consists of the following components:

evalScanner.c – The main function. This program and the test files are provided for you on Brightspace in a separate file (**Assignment 2.zip**).

token.h – Provided **complete**. It contains the declarations and definitions describing different tokens. Do not modify the declarations and the definitions. **You should not change this file**.

table.h – Provided **incomplete**. It contains transition table declarations necessary for the scanner. All of them are incomplete. You must initialize them with proper values. It must also contain the function prototypes for the accepting functions. You are to complete this file. You will find the additional requirements within the file. If you need named constants, you can add them to that file.

scanner.c - Provided **incomplete**. It contains a few declarations and definitions necessary for the scanner. You will find the additional requirements within the file.

- The definition of the ***startScanner()*** is complete and you must not modify it. The function performs the initialization of the scanner input buffer and some other scanner components.
- You need to write the function ***tokenizer()*** which performs the token recognition (*the original idea is from the concept:* **m**atch-**a**-**l**exeme-**a**nd-**r**eturn = "*malar*").
    - It "reads" the lexeme from the input stream (in our case from the input buffer) one character at a time and returns a token structure any time it finds a token pattern (as defined in the lexical grammar) which matches the lexeme found in the stream of input symbols.

- The token structure contains the **token code** and the **token attribute**.
- The token attribute can be an attribute code, an integer value, a floating-point value (for the floating-point literals), a lexeme (for the variable identifiers and the errors), an offset (for the string literals), an index (for the keywords), or source-end-of file value.
- Remember that:
  - The scanner ignores the white space.
  - The scanner ignores the comments as well. It ignores all the symbols of the comment including line terminator.
- The function consists of **two implementation parts** (see section 2.1 from this document).
  - Part 1: token-driven (special case or exception-driven) processing.
  - Part 2: transition-table-driven processing.
- You need to write both parts. The tokens which must be processed one by one (special cases or exceptions) are defined in **table.h**.
- **Note**: You must build the transition table for recognizing the variable identifiers (including keywords), integer literals, floating-point literals, and string literals.

Note 2: Progressive assignment

Remember that you need to use the code previously developed in your buffer:

** **buffer.h**: Completed in Assignment 1. It contains buffer structure declarations, as well as function prototypes for the buffer structure.*

** **buffer.c**: Completed in Assignment 1. It contains the function definitions for the functions written in Assignment 1.*

***Note**: some functions in buffer were note used in Assignment 1, but will be required to finish Assignment 2.*

The scanner is to perform some **rudimentary error handling** – **error detection** and **error recovery**.

- ***Error handling in comments***. If the **comment construct** is not lexically correct (as defined in the grammar), the scanner must return an error token.
  - For example, if the scanner finds the symbol # but the symbol is not followed by the symbol # it must return an **error token**.
  - The attribute of the comment error token is a **C-type string** containing the # symbol and the symbol following # (##).

  - Before returning the error token, the scanner **must ignore** all of the symbols of the wrong comment to the end of the line.

- ***Error handling in strings*****.** In the case of **illegal strings**, the scanner must return an error token.
  - The erroneous string must be stored as a C-type string in the attribute part of the error token (***not in the string literal table***).
  - If the erroneous string is longer than 20 characters, you must store the first 17 characters only and then append three dots (**…**) at the end.
- ***Error handling in case of illegal symbols*****.** If the scanner finds an **illegal symbol** (out of context or not defined in the language alphabet) it returns an error token with the erroneous symbol stored as a C-type string in the attribute part of the token.
- ***Error handling of runtime errors*****.** In a case of **run-time error**, the function must store a non-negative number into the global variable **errorNumber** and return a run-time error token. The error token attribute must be the string "RUN TIME ERROR:".
  - **TIP_1:** It is important to be sure that you are incrementing the line number when you find a new line char in your buffer.
- The definition of the ***nextState()*** is **complete** and you must not modify it.
- The function ***nextClass()*** is **incomplete** and you must return the **column index** for the column in the transition table that represents a character or character class / type.
  - For example, the representation for letters in the RE (Regular Expression) is L = [a - z A - Z] and must return the "0" because the order of TT (see **CST8152_RETDTT_W21.doc**).
- Additionally, you have to write the definitions of the **accepting functions** and some other functions (see **scanner.c**).
  - Remember that you need to **accept** (recognize) the tokens defined as AVID, SVID, IL, FPL, SL:
    - Token ***funcAVID*** (char* lexeme);
    - Token ***funcSVID*** (char* lexeme);
    - Token ***funcIL*** (char* lexeme);
    - Token ***funcFPL*** (char* lexeme);
    - Token ***funcSL*** (char* lexeme);
  - You must also create the function to set the Error Token (when DFA is not finishing with the correct token recognition):
    - Token ***funcErr*** (char* lexeme);
  - **Note:** You may implement your own functions if needed.

Note 3: Spec violation manipulating Buffer

In the scanner implementation you are not allowed to manipulate directly any of the Buffer structure data members. You must use appropriate functions provided by the buffer implementation. Direct manipulation of data members will be considered an error against the functional specifications and will render your Scanner non-working.

2.3. IMPLEMENTATION STEPS

1. Firstly, be sure your **buffer** (Assignment 1) is working fine (at least for standard tests).
    a. *Problems with buffer will directly affect your next assignments*.
    b. *However, it is possible to start Assignment 2 in parallel, focusing on the Task 1 (models for PLATYPUS).*

- ***IMPORTANT NOTE_1:*** The answer for buffer is not provided and it is required that the student / team has developed it previously.

2. Start Task 1 (**5 marks**):
    a. It is required to answer the questions in **CST8152_RETDTT_W21** *before starting the development* of **Task 2**.
    b. To do this, read the language specification – both informal and BNF that you can find respectively on **PlatypusILS_W21** and **PlatypusBNFGR_W21**.
3. Start Task 2 (**15 marks**): Complete all "**TODO**" sections in your files:
    a. On **table.h**:
        i. Define the constants, the elements (transitionTable, stateType, finalStateTable) and headers for functions to be implemented.
        ii. **TODO_101:** Follow the standard and adjust the file header.
        iii. **TODO_102:** Following PLATYPUS spec define constants for EOF (two situations must be considered).
        iv. **TODO_103:** Define constants for Token Errors and illegal state;
        v. **TODO_104:** Define values missing on transitionTable;
            - **TIP:** Check the transition table developed in the Task 1.
        vi. **TODO_105:** Define values for accepting states types;
        vii. **TODO_106:** Define list of acceptable states;

            To do this, consider that the numbers for **ASWR** or **ASNR** (because **NOAS** are already done). for the final states. They will be required to complete the stateType table.

viii. **TODO_107:** Declare accepting states functions

- **TIP:** see the function names defined in the accepting functions.

To do this, include all the function definitions that return Token when receiving a specific lexeme – they are already started in the buffer.c (for instance, Token FuncName(char lexeme[])), responsible to recognize AVID, SVID, IL, FPL, SL.

ix. **TODO_108:** Define finalStateTable.

x. **TODO_109:** Define the number of Keywords from the language.

xi. **TODO_110:** Define the keywordTable.

b. On **scanner.c**: continue the development:

i. **TODO_201:** Follow the standard and adjust the file header.

ii. **TODO_202:** Follow the standard and adjust all the function headers.

iii. **TODO_203:** Adjust the tokenizer() header;

iv. Implement the two parts of tokenizer(): token driver scanner (where you are detecting some terminals) and transition table driver scanner (where you call the FDA functions.

- **TODO_204:** processToken() part 1: Token driven scanner implementation using switch

THE PART 1 OF PROCESS TOKEN IS SUPPOSED IS BASICALLY THE switch CASE WHEN YOU ARE READING CHARS.
A. SEVERAL TIMES, WITH ONE SINGLE CHAR, YOU CAN DECIDE ABOUT THE TOKEN TO BE CLASSIFIED: IT CAN BE DIRECTLY (FOR INSTANCE, '**(**').
B. BUT IN OTHER CASES, IT IS NECESSARY TO READ A SEQUENCE OF CHARS (EX: '**.**','**O**','**R**','**.**' TO MATCH WITH THE TOKEN "**.OR.**"). YOU CAN USE **bSetMarkOffset()** TO THIS.
C. YOU MUST ADJUST THE code (IN **currentToken**).
D. IN SOME CASES, ADDITIONAL INFO (FOR INSTANCE, attribute FIELD) SHOULD BE UPDATED BECAUSE YOU NEED TO SPECIFY WHAT IS THE VALUE FOR UNION THAT YOU ARE USING.
E. REMEMBER TO ADJUST THE LINE (GLOBAL VARIABLE **line**) WHEN NECESSARY.
F. CONSIDER ERROR SITUATIONS AND USE DEFENSIVE CODE.

- **TODO_205:** processToken() part 2: Transition driven scanner implementation inside default

THE PART 2 OF PROCESS TOKEN IS SUPPOSED TO HAPPEN IN THE "**default**" CASE OF THE SWITCH. IN THIS PART, WE WILL USE SEVERAL VARIABLES THAT ARE DECLARED IN THE BEGINNING: (**state**, **lexStart**, **lexEnd**, **lexLength**, **lexemeBuffer**) AS WELL AS SOME BUFFER FUNCTIONS THAT WERE NOT USED (YET) IN THE 1st ASSIGNMENT, BUT SHOULD BE IMPLEMENTED.
A. USE THE **state** TO GET THE NEXT STATE (CALLING **nextState()**)
B. USE **lexStart** TO GET THE INITIAL POSITION OF THE LEXEME (USING **bGetChOffset()**)
C. YOU MUST MARK THIS POSITION CALLING **bSetMarkOffset()**
D. NOW, IT IS TIME TO CREATE A LOOP THAT WILL STOP UNTIL YOU HAVE FOUND AN ACCEPTABLE STATE: SO USE THE "**NOAS**" TO REPEAT THE PROCESS:
  D.1. GETTING THE NEXT CHAR
  D.2. GETTING THE NEXT STATE
E. WHEN YOU HAVE FOUND A FINAL STATE, IF IT IS "**ASWR**", YOU NEED TO RETRACT – CALLING **bRetract()**..
F. SET THE lexEnd AND CALCULATE THE LENGTH OF LEXEME (USING **lexLength**)
G. YOU NEED TO CREATE A TEMPORARY BUFFER TO LEXEME (CALLING **bCreate()** in **FIXMODE**, USING THE APPROPRIATE SIZE)
H. RESET THE BUFFER (CALLING **bRestore()**) TO MOVE TO THE SPECIFIC MARK POSITION.
I. COPY THE LEXEME TO **lexemeBuffer**.

J. CALL THE APPROPRIATE FUNCTION TO RETURN THE TOKEN BY USING **finalStateTable** AND USING THE **lexemeBuffer**
K. RETURN THE **currentToken**
L. CONSIDER ERROR SITUATIONS AND USE DEFENSIVE CODE.

v. Adjust your **nextClass**(), to identify the column in the TT:
   - **TODO_206:** adjusting the header.
   - **TODO_207:** the logic to return the next column in TT

vi. Implement all functions required to recognize the tokens, starting with ***funcAVID()***:
   - **TODO_208:** adjusting the header.
   - **TODO_209:** the logic to recognize **AVID** (arithmetic variable identifier)

- Remember to respect the limit defined for lexemes (VID_LEN) and set the lexeme to the corresponding attribute (vidLexeme).
- Remember to end each token with the '\0'.
- Suggestion: Use "strncpy" function.

vii. Implement now the ***funcSVID()*** that recognizes string VID.
   - **TODO_210:** adjusting the header.
   - **TODO_211:** the logic to recognize **SVID**:

- SVID must start and end with $.
- Remember to respect the limit defined for lexemes (VID_LEN), also observing the delimiters. Remember to end also with \0.
- Set the lexeme to the corresponding attribute (vidLexeme).
- Suggestion: Use "strncpy" function.

viii. Continue with ***funcIL()***, responsible for recognizing Integer Literals.
   - **TODO_212:** adjusting the header.
   - **TODO_213:** the logic to recognize **IL**:

- It is necessary respect the limit (ex: 2-byte integer in C).
- In the case of larger lexemes, error should be returned.
- Only first ERR_LEN characters are accepted and eventually, additional three dots (...) should be put in the output.

ix. Continue with ***funcFPL()***, that recognizes Float Point Literals.
   - **TODO_214:** adjusting the header.
   - **TODO_215:** the logic to recognize **FPL**:

- It is necessary respect the limit (ex: 4-byte integer in C).
- In the case of larger lexemes, error should be returned.
- Only first ERR_LEN characters are accepted and eventually, additional three dots (...) should be put in the output.

x. Continue with ***funcSL()***, that recognizes String Literals.
   - **TODO_216:** adjusting the header.

- **TODO_217:** the logic to recognize **SL**:

- The lexeme must be stored in the String Literal Table (**stringLiteralTable**).
- You need to include the literals in this structure, using offsets.
- Remember to include '\0' to separate the lexemes.
- Remember also to increment line if necessary.

xi. Now, continue with ***funcErr()***, that deals with errors.

- **TODO_218:** adjusting the header.
- **TODO_219:** the logic to adjust error messages:

- This function uses the **errLexeme**, respecting the limit given by ERR_LEN. If necessary, use three dots (...) to use the limit defined.
- The error lexeme contains line terminators, so remember to increment line, if necessary.

xii. Continue now the implementation: adjust the ***isKeyword()***, that identifies the corresponding keyword from language (if there is).

- **TODO_220:** adjusting the header.
- **TODO_221:** complete the code.

xiii. Finally, if necessary, create additional functions (remember to include definition in table.h):

- **TODO_222:** (if necessary).

4. Finally, start testing with the files (see files on **Assignment .zip**).:
   a. The sequence suggested to your tests is:
      i. a201empty.pls: Must match with a201empty.sout
      ii. a202r.pls: Must match with a202r.sout
      iii. a203w.pls: Must match with a203w.sout
      iv. a204error.pls: Must match with a204error.sout
   b. Test also the additional test files to check the corresponding outputs: a205line.pls, a206coml.pls, a207strl.pls, a208nl.pls, a209esl.pls, a210lint.pls, a211comro.pls, a212t7.pls, a213dot.pls, a214com.pls, a215dAE.pls, a216dnl.pls, a217err.pls, a218kw.pls, a219test.pls, a220test.pls.

- **TIP_3: Check all comments included in the files (.h and .c) that you are downloading.**

Submission Details

❖ **Digital Submission:** Here are the general orientation. Any problems, contact your lab professor:

- **Compress** into a **zip** file all the files used in the project (for instance, **buffer.h**, **buffer.c**, **table.h**, **token.h**. **scanner.c**, **evalScanner.c** as well as the **files required** in the Marking Sheet for A1 (for instance, the document with **answers from the Part 1** – RE, TD and TT)
  - Any **additional files** related to project- your additional input/output test files if you have any can be included (not required).
- Please check the documentation required (as shown in **CST8152_A2MarkingSheet_W21**):
  - For instance, a **Cover page** and a **Test Plan**. Check the Assignment 1 Marking Sheet for it, as well as Submission Standard document.

- The submission must follow the course **submission standards**. You will find the Assignment Submission Standard (**CST8152_ASSAMG_W21)** for the Compilers course on the Brightspace.
- **Upload** the zip file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.
- ***IMPORTANT NOTE_2:*** The name of the file must be **Your Last Name** followed by the last three digits of your student number followed by your lab **section number**. For example: **Sousa123_s10.zip**.
  - If you are working in teams, please, include also your partner info. For instance, something like: **Sousa123_Melo456_s10.zip**.
  - **Remember:** Only students from the same section can constitute a specific team.

Note 4: About Teams

*You can submit individually or in teams (2 students only). This team can be different from the previous assignment, but it is required to inform your lab professor previously. In this case, it is required to submit one page detailing who was responsible for which function (as required in the header function). Some methods, such as **tokenizer()** can have two authors, but NOT all methods. For this reason, it is possible to find different marks for each student even in the same team, when the assignment is evaluated.*

- ***IMPORTANT NOTE_3:*** **Assignments will not be marked if there are not source files in the digital submission.** Assignments could be late, but the lateness will affect negatively your mark: see the Course Outline and the Marking Guide. All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

- **Evaluation Note:** Make your functions as efficient as possible.

- If your program compiles, runs, and produces correct output files, it will be considered a ***working program***.
- Additionally, I will try my best to "crash" your functions using a modified main program, which will test all your functions including calling them with "invalid" parameters.
- I will use also some additional test files (for example, a large file). So, test your code as much as you can!
- This can lead to fairly big reduction of your assignment mark (see **CST8152_ASSAMG** and **MarkingSheetA2** documents).

❖ ***IMPORTANT NOTE_4:*** In case of emergency (BS LMS is not working) submit your zip file via e-mail to your **lab professor**.

About Lab Demos

❖ **Main Idea:** This semester, you can get **bonuses** when you are demonstrating your assignment during the lab. The marks are reported in CSI.

- **Note:** The demo during lab sessions is now required to get marks when you do your lab submissions.

❖ **How to Proceed:** You need to demonstrate the expected portion of code to your Lab Professor in **private Zoom Sections**.

- If you are working in teams, you and your partner must do it together, otherwise, only the student that has presented can get the bonus marks.
- Eventual questions can be posed by the Lab professor for any explanation about the code developed.
- Each demo is related to a **specific lab** in one specific week. If it is not presented, no marks will be given later (even if the activity has been done).

Finally, another motivation thought that prof **Svillen Ranev** used to share…

"There are two kinds of people, those who do the work and those who take the credit. Try to be in the first group; there is less competition there."

Indira Gandhi

File update: Jan 1st 2021.

Good Luck with Assignment 2!